Inteligência Computacional – 2023/24

Projeto – Fase2

Elementos do grupo:

Luís Henrique P. O. Travassos, nº2021136600

Rodrigo Ramalho Ferreira, nº2021139149

Conteúdo:

1. Em que consiste a Computação Swarm

A Computação em Enxame, inspirada nas complexas dinâmicas colaborativas de enxames biológicos, como formigas e abelhas, representa um paradigma inovador no campo da computação. Este conceito aplica-se à interação e colaboração descentralizada entre múltiplos agentes autónomos, sejam computadores, robôs ou sensores, que, apesar de seguirem regras simples e operarem sem um controle centralizado, são capazes de gerar comportamentos coletivos sofisticados e adaptativos. Estes agentes, ao compartilharem informações e coordenarem esforços, conseguem não só adaptar-se a mudanças ambientais, mas também resolver problemas complexos de maneira eficiente e resiliente.

Na esfera da inteligência artificial, e mais especificamente no treino de redes neurais artificiais, que são componentes essenciais da aprendizagem de máquina, a Computação em Enxame tem se mostrado particularmente benéfica. O treino destas redes é uma tarefa que exige grande capacidade computacional e pode ser extremamente demorado. Através da aplicação dos princípios de Computação em Enxame, é possível não só acelerar este processo, mas também melhorar a qualidade do treino.

Isto acontece devido à capacidade desta abordagem em realizar a otimização de hiperparâmetros, ajustando variáveis críticas como taxas de aprendizagem e arquiteturas de rede, e ao seu potencial em facilitar o treino distribuído. No treino distribuído, diferentes agentes trabalham em partes distintas da rede neural de forma independente, para depois combinarem seus resultados, melhorando a eficiência e reduzindo o tempo necessário para o treino.

Além disso, a Computação em Enxame tem mostrado resultados promissores na deteção de anomalias em grandes conjuntos de dados, uma tarefa particularmente desafiadora devido à necessidade de identificar padrões não evidentes que podem indicar irregularidades ou novas oportunidades de negócio. Esta capacidade de detetar o indetetável, por assim dizer, torna a Computação em Enxame uma ferramenta valiosa para empresas e organizações que dependem do processamento e análise de grandes volumes de dados.

A sinergia entre a Computação em Enxame e o treino de redes neurais reflete a tendência crescente da computação em imitar processos naturais para resolver problemas tecnológicos complexos. Ao emular a inteligência coletiva de enxames naturais, esta abordagem proporciona uma nova dimensão de adaptabilidade e eficiência em campos que vão desde a análise de dados até a inteligência artificial, passando pela otimização de processos e a robótica. À medida que avançamos na compreensão e implementação deste paradigma, podemos esperar uma expansão das suas aplicações e uma contribuição ainda maior para o progresso tecnológico.

2. Análise e Comparação entre o algoritmo GWO e PSO

O algoritmo de Otimização de Lobos Cinzentos (Gray Wolf Optimization - GWO) é uma técnica de otimização bio-inspirada que simula o comportamento social e de caça dos lobos cinzentos. Este algoritmo foi desenvolvido para resolver problemas de otimização complexos, adaptando as estratégias colaborativas dos lobos na natureza.

No GWO, a população de lobos é dividida em quatro grupos, refletindo a hierarquia social dos lobos: o alfa (o líder), o beta (o segundo no comando), o delta (lobos subordinados) e os ómegas (os últimos na hierarquia). O algoritmo começa com a inicialização de uma população de lobos cinzentos (soluções candidatas) distribuídos aleatoriamente pelo espaço de busca. O alfa, beta e delta são as três melhores soluções após a avaliação da função objetivo.

O processo de caça (busca pela solução ótima) é guiado principalmente pelo alfa, seguido pelo beta e pelo delta. Os lobos ómega seguem esses líderes. A localização da presa (solução ótima) é desconhecida, mas os lobos vão se aproximando dela através da atualização dinâmica das suas posições nas iterações do algoritmo, que são influenciadas pelas posições do alfa, beta e delta.

As posições dos lobos são atualizadas utilizando equações que consideram a distância entre eles e os líderes, bem como a capacidade de cada lobo de se aproximar ou afastar-se da presa, o que reflete o cerco e o ataque dos lobos reais durante a caça.

Comparativamente ao PSO (Particle Swarm Optimization), que também é um algoritmo de otimização baseado no comportamento social – neste caso, de bandos de pássaros ou cardumes de peixes –, o GWO oferece uma representação mais complexa da hierarquia social e das interações entre os indivíduos. Enquanto o PSO ajusta as trajetórias das partículas (soluções) com base na experiência própria e dos vizinhos, o GWO utiliza a hierarquia dos lobos para influenciar a adaptação das soluções.

Vantagens do GWO em relação ao PSO incluem:

- Estrutura social hierárquica que pode evitar a convergência prematura para mínimos locais.
- Menor número de parâmetros a ajustar (no PSO, deve-se definir a inércia, o fator cognitivo e o fator social).
- Pode ser mais eficaz em espaços de busca complexos devido à sua estratégia de atualização de posições.

Desvantagens:

- Pode ser mais computacionalmente intensivo devido à complexidade das interações sociais.
- A definição da hierarquia pode limitar a diversidade de soluções se os líderes convergirem rapidamente para uma solução subótima.
- O GWO pode ter dificuldades em sair de mínimos locais se os três melhores lobos estiverem posicionados próximos a um.

3. Análise da influência do algoritmo PSO na função de Ackley

3.1. Demonstração código

```python
import numpy as np
import pyswarms as ps
```

```python
# Definição da função de Ackley
def ackley_function(x):
    a = 20
    b = 0.2
    c = 2 * np.pi
    d = len(x)  # Dimensão do vetor de entrada
    sum_sq_term = -a * np.exp(-b * np.sqrt(np.sum(x**2) / d))
    cos_term = -np.exp(np.sum(np.cos(c * x) / d))
    result = sum_sq_term + cos_term + a + np.exp(1)
    return result
```

```python
# Definindo os limites do espaço de busca para a função de Ackley
# Estes são os limites comuns para testar a função de Ackley
bounds = (np.array([-32.768] * 2), np.array([32.768] * 2))  # Ajuste para 2 dimensões

# Definindo os limites do espaço de busca para a função de Ackley para 3 dimensões
#bounds = (np.array([-32.768] * 3), np.array([32.768] * 3))  # Ajuste para 3 dimensões

# Configurações do PSO
options = {'c1': 0.5, 'c2': 0.3, 'w': 0.9}

# Inicializando o PSO
optimizer = ps.single.GlobalBestPSO(n_particles=30,
                                    dimensions=2,
                                    options=options,
                                    bounds=bounds)
```

```python
# Executando a otimização
cost, pos = optimizer.optimize(ackley_function, iters=100)

# Imprimindo o resultado da otimização
print(f"Melhor custo encontrado: {cost}")
print(f"Melhor posição encontrada: {pos}")
```

Este código implementa o algoritmo de Otimização por Enxame de Partículas (PSO) para encontrar o mínimo global da função de Ackley, que é uma função complexa e desafiadora com vários mínimos locais.

O objetivo é minimizar o "custo" calculado pela função, baseado na posição das partículas no espaço de busca, que pode ser de duas ou três dimensões. O PSO usa parâmetros como confiança pessoal (c1), confiança social (c2) e inércia (w) para equilibrar a exploração e exploração do espaço de busca. Inicialmente, 30 partículas são utilizadas em 100 iterações.

O resultado mostra a eficácia do PSO em alcançar o mínimo global da função, que é 0 nas coordenadas (0,0) para o espaço bidimensional ou (0,0,0) para o tridimensional.

3.2. Análise dos diferentes parâmetros

Parâmetros	Melhor Custo (2 Dim.)	Melhor Posição (2 Dim.)	Melhor Custo (3 Dim.)	Melhor Posição (3 Dim.)	Objetivo
c1 = 0.5 **c2 = 0.3** **w = 0.9** **n_particles = 30** **iters = 100**	21.13	(-23.51, 24.94)	20.96	(-26.13, 21.90, 25.31)	Valores iniciais e escolhidos aleatoriamente, só para servir de início à análise.

c1 = 1.0 **c2 = 1.0** **w = 0.9** **n_particles = 30** **iters = 100**	17.49	(4.58, -2.85)	20.69	(4.79, -20.04, 17.33)	Avaliar o impacto de um maior equilíbrio entre a componente cognitiva e a componente social no desempenho do algoritmo.
c1 = 0.5 **c2 = 0.3** **w = 0.4** **n_particles = 30** **iters = 100**	21.00	(-26.95, -6.71)	20.96	(-6.49, 19.90, 17.51)	Observar como uma menor inércia (que promove uma maior exploração do espaço de busca) afeta a capacidade do algoritmo de encontrar o mínimo global.
c1 = 0.5 **c2 = 0.3** **w = 0.9** **n_particles = 100** **iters = 100**	20.26	(-15.89, -3.66)	17.76	(-23.32, 4.08, 2.01)	Determinar se um enxame maior melhora a convergência para o mínimo global devido a uma maior diversidade de soluções sendo exploradas.
c1 = 0.5 **c2 = 0.3** **w = 0.9** **n_particles = 30** **iters = 50**	21.22	(-7.25, 30.44)	20.96	(10.30, 20.47, -0.92)	Verificar se o algoritmo ainda consegue encontrar uma boa solução com menos iterações.
c1 = 1.0 **c2 = 1.0** **w = 0.9** **n_particles = 100** **iters = 100**	21.23	(-17.02, -28.88)	21.39	(-22.39, -30.75, -0.51)	Com a análise dos resultados anteriores iremos, como análise final, aumentar os valores de c1, c2 e n_particles, enquanto mantemos o valor de w e iters.

Através de uma série de testes com o algoritmo PSO aplicado à função de Ackley, foi possível observar o impacto que diferentes configurações de parâmetros têm sobre a eficácia do algoritmo em encontrar o mínimo global.

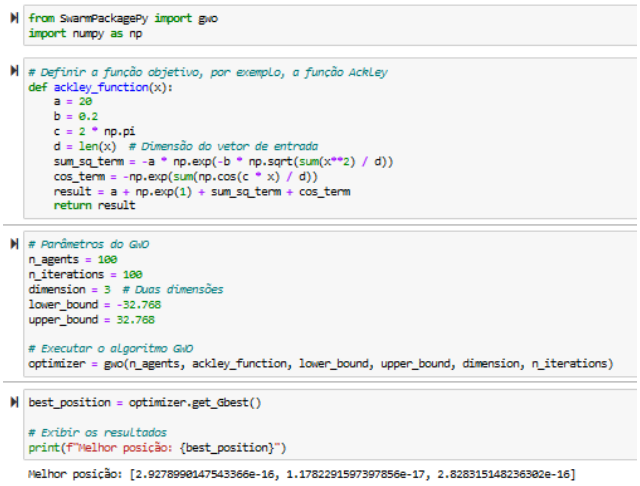
Notou-se que aumentos em c1 e c2, que promovem uma maior influência das componentes cognitiva e social, respectivamente, e um maior número de partículas tendem a melhorar a precisão da busca pelo mínimo global, tanto em duas quanto em três dimensões.

Contudo, mesmo com essas alterações, o mínimo global conhecido não foi atingido, o que sugere um espaço para aprimoramento na escolha de parâmetros ou na própria metodologia do algoritmo.

Mantendo o valor de w e iters constantes, a análise final mostra um compromisso entre exploração do espaço de busca e a convergência rápida, evidenciando a complexidade e a necessidade de um ajuste fino nas configurações do PSO para otimização eficaz em diferentes problemas e dimensões.

3.3. Análise dos resultados do GWO com os "melhores" parâmetros

Código do algoritmo GWO com a função Ackley:



```
from SwarmPackagePy import gwo
import numpy as np

# Definir a função objetivo, por exemplo, a função Ackley
def ackley_function(x):
    a = 20
    b = 0.2
    c = 2 * np.pi
    d = len(x)  # Dimensão do vetor de entrada
    sum_sq_term = -a * np.exp(-b * np.sqrt(sum(x**2) / d))
    cos_term = -np.exp(sum(np.cos(c * x) / d))
    result = a + np.exp(1) + sum_sq_term + cos_term
    return result

# Parâmetros do GWO
n_agents = 100
n_iterations = 100
dimension = 3  # Duas dimensões
lower_bound = -32.768
upper_bound = 32.768

# Executar o algoritmo GWO
optimizer = gwo(n_agents, ackley_function, lower_bound, upper_bound, dimension, n_iterations)

best_position = optimizer.get_gbest()

# Exibir os resultados
print(f"Melhor posição: {best_position}")
```

```
Melhor posição: [2.9278990147543366e-16, 1.1782291597397856e-17, 2.828315148236302e-16]
```

Análise dos resultados com os melhores parâmetros do algoritmo GWO:

Parâmetros	**Melhor Posição (2 Dim.)**	**Melhor Posição (3 Dim.)**	**Objetivo**
n_agents = 100 **n_iterations = 100**	(-3.90, -1.48)	(2.93, 1.18, 2.83)	Comparar os resultados obtidos pelo algoritmo PSO com o algoritmo GWO

Com o uso dos "melhores" valores para os parâmetros do algoritmo PSO no algoritmo GWO podemos chegar à conclusão que apesar do GWO possuir menos parâmetros de configuração ainda assim aproxima-se muito mais do resultado pretendido, neste caso o mínimo global (0, 0) ou (0, 0, 0), ao contrário do algoritmo PSO, podendo assim afirmar-se, necessitando de mais experimentos, de que este algoritmo têm um melhor desempenho.

4. Comparação do algoritmo GWO e PSO na otimização da arquitetura

4.1. Análise do cenário de testes

No contexto atual de testes, enfrentamos um cenário desafiador ao empregar algoritmos de otimização avançados como o Gray Wolf Optimization (GWO) e o Particle Swarm Optimization (PSO). O dataset em questão é notoriamente pesado, o que introduz uma dificuldade significativa para a utilização eficiente destas técnicas. A grande dimensão e complexidade dos dados resultam em um tempo extenso para executar até mesmo uma única execução do código, limitando fortemente a nossa capacidade de testar e iterar. (início -> 11:43, fim -> 16:48)

Com apenas dois hiperparâmetros disponíveis para ajuste, a taxa de aprendizagem (learning-rate) e a taxa de desativação (dropout-rate), as nossas opções de experimentação são reduzidas. Esses parâmetros são cruciais pois influenciam diretamente a eficiência e a eficácia do processo de

aprendizagem do modelo. No entanto, a restrição de tempo impede uma exploração mais aprofundada do espaço de hiperparâmetros.

Considerando a implementação atual com 5 partículas e 5 iterações, temos uma amostragem bastante limitada, que não permite uma avaliação extensiva da performance dos algoritmos. Idealmente, um número maior de partículas e iterações conduziria a uma otimização mais refinada e conclusões mais robustas. No entanto, a realidade do dataset pesado e do tempo de execução proibitivo nos obriga a aceitar estas limitações.

A consequência direta desta situação é que as conclusões tiradas dos testes atuais devem ser consideradas com cautela. Embora os resultados possam fornecer uma indicação inicial sobre o comportamento dos algoritmos e a influência dos hiperparâmetros, eles estão longe de serem conclusivos. A incapacidade de realizar um número maior de testes ou de ajustar os hiperparâmetros com mais granularidade significa que qualquer insight obtido deve ser validado com dados adicionais ou em condições experimentais mais controladas, quando possível.

Em resumo, a pesada carga computacional do dataset impõe restrições significativas à aplicação de GWO e PSO, limitando a otimização aos parâmetros disponíveis e ao número de iterações práticas. Portanto, a análise dos resultados requer uma abordagem conservadora, reconhecendo as limitações e a necessidade de testes futuros mais extensivos.

4.2. Apresentação Código PSO (Particle Swarm Optimization)

O código apresentado descreve o processo de usar um algoritmo de Otimização por Enxame de Partículas (PSO) para treinar uma rede neural convolucional (CNN) e otimizar seus hiperparâmetros - taxa de aprendizagem e taxa de desativação (dropout).

A função train_network define e compila o modelo da CNN, treina-o com um conjunto de dados de treino, e avalia-o utilizando um conjunto de validação para obter a perda de validação, que serve como uma medida de quão bem o modelo está a funcionar.

A função fitness_function é usada pelo PSO para avaliar a "adequação" de cada conjunto de hiperparâmetros, executando train_network para cada partícula (conjunto de hiperparâmetros) e retornando as perdas de validação correspondentes.

O otimizador PSO é inicializado com limites para a taxa de aprendizagem e a taxa de desativação, e executa 5 iterações para encontrar o melhor conjunto de hiperparâmetros que minimiza a perda de validação.

Finalmente, o modelo final é treinado com a melhor taxa de aprendizagem e taxa de desativação encontradas, e o histórico de treino é registado, o que pode ser usado para análise posterior do desempenho do modelo.

```python
# Defina as funções de treinamento de rede e fitness para o PSO
def train_network(hyperparameters, train_x, train_y, val_x, val_y):

    learning_rate, dropout_rate = hyperparameters

    # Definindo o modelo da rede neural convolucional
    model = Sequential([
        Conv2D(64, (3, 3), activation='relu', input_shape=(im_size, im_size, 3)),
        MaxPooling2D((2, 2)),

        Flatten(),

        Dropout(dropout_rate),

        Dense(len(street_types), activation='softmax')
    ])

    model.compile(optimizer=Adam(learning_rate=learning_rate),
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])

    model.fit(train_x,
              train_y,
              epochs=5,
              validation_data=(val_x, val_y),
              verbose=0)

    validation_loss, validation_accuracy = model.evaluate(val_x, val_y, verbose=0)

    return validation_loss
```

```python
def fitness_function(x, train_x, train_y, val_x, val_y):

    n_particles = x.shape[0]

    losses = []

    for i in range(n_particles):
        # x[i] contém os hiperparâmetros para a i-ésima partícula
        hyperparameters = x[i]
        loss = train_network(hyperparameters, train_x, train_y, val_x, val_y)
        losses.append(loss)

    return np.array(losses)
```

```python
# Inicializando o otimizador GlobalBestPSO do pyswarms
bounds = [(0.0001, 0.1), (0.0, 0.5)]  # (min_learning_rate, max_learning_rate), (min_dropout_rate, max_dropout_rate)
options = {'c1': 0.5, 'c2': 0.3, 'w': 0.9}

optimizer = ps.single.GlobalBestPSO(n_particles=5,
                                    dimensions=2,
                                    options=options,
                                    bounds=bounds)

# Executando o PSO para encontrar os melhores hiperparâmetros
cost, best_pos = optimizer.optimize(fitness_function,
                                    iters=5,
                                    train_x=train_x,
                                    train_y=train_y,
                                    val_x=val_x,
                                    val_y=val_y)

best_learning_rate, best_dropout_rate = best_pos
```

```
2023-11-08 11:44:11,151 - pyswarms.single.global_best - INFO - Optimize for 5 iters with {'c1': 0.5, 'c2': 0.3, 'w': 0.9}
pyswarms.single.global_best: 100%|██████████|5/5, best_cost=0.196
2023-11-08 16:33:36,606 - pyswarms.single.global_best - INFO - Optimization finished | best cost: 0.1959879696369171, bes
t pos: [0.00018259 0.10997558]
```

```python
final_model = Sequential([
    Conv2D(64, (3, 3), activation='relu', input_shape=(im_size, im_size, 3)),
    MaxPooling2D((2, 2)),

    Flatten(),

    Dropout(best_dropout_rate),

    Dense(len(street_types), activation='softmax')
])

final_model.compile(optimizer=Adam(best_learning_rate),
                    loss='sparse_categorical_crossentropy',
                    metrics=['accuracy'])

history = final_model.fit(train_x,
                          train_y,
                          epochs=5,
                          validation_data=(val_x, val_y))
```

4.3. Análise de resultados (Particle Swarm Optimization)

```
Best learning rate: 0.00018258859132538424
Best dropout rate: 0.10997558109383135

Test accuracy: 92.50%

Confusion Matrix:
[[176  20   4]
 [  8 192   0]
 [ 12   1 187]]

Classification Report:
              precision    recall  f1-score   support

       clean       0.90      0.88      0.89       200
      litter       0.90      0.96      0.93       200
     recycle       0.98      0.94      0.96       200

    accuracy                           0.93       600
   macro avg       0.93      0.92      0.93       600
weighted avg       0.93      0.93      0.93       600


AUC for class clean: 0.98
AUC for class litter: 0.99
AUC for class recycle: 0.99
```

Os resultados da otimização com o algoritmo PSO indicam que a melhor taxa de aprendizagem encontrada foi de aproximadamente 0.00018, e a melhor taxa de desativação (dropout) foi de aproximadamente 0.11. Estes hiperparâmetros contribuíram para uma precisão de teste do modelo de 92,50%, o que é um resultado bastante alto.

Analisando a matriz de confusão, podemos ver que a classe 'clean' teve algumas confusões com as outras categorias, enquanto 'recycle' teve o melhor desempenho com a maior precisão. A precisão (precision) e a recuperação (recall) para 'clean' e 'litter' são semelhantes, ambas com valores em torno de 0.90, indicando um equilíbrio entre a capacidade do modelo de identificar corretamente as classes e de cobrir todos os exemplos reais daquelas classes. 'Recycle' teve uma precisão excepcionalmente alta e um bom recall, resultando num F1-score impressionante de 0.96.

O relatório de classificação fornece uma visão detalhada do desempenho do modelo para cada classe, e todos os valores de F1-score são altos, indicando um desempenho geral robusto do modelo.

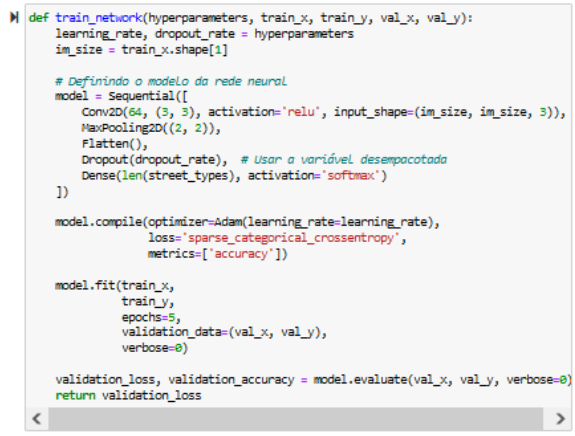
4.4. Apresentação Código GWO (Gray Wolf Optimization)

O código define duas funções principais para treinar uma rede neural e para servir como função de fitness em um processo de otimização. A função train_network aceita hiperparâmetros - taxa de aprendizagem e taxa de desativação (dropout) - e utiliza-os para construir e treinar uma rede neural convolucional (CNN) com dados de treino e validação, retornando a perda de validação como um indicador de desempenho do modelo.

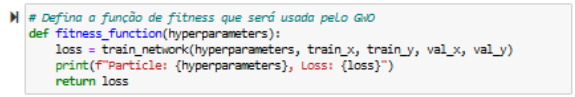
A função fitness_function é utilizada pelo algoritmo de otimização Gray Wolf Optimization (GWO) para avaliar a performance de diferentes conjuntos de hiperparâmetros, imprimindo a perda associada a cada conjunto.

O otimizador GWO é configurado com um número definido de agentes e iterações, assim como limites superiores e inferiores para os hiperparâmetros. Ele é executado para encontrar o melhor conjunto de hiperparâmetros que minimiza a perda de validação.

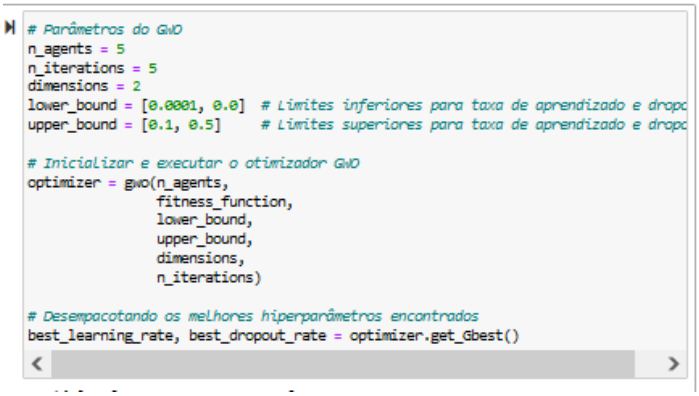
Por fim, os melhores hiperparâmetros encontrados pelo GWO são utilizados para compilar e treinar o modelo final, após o qual o histórico de treino é obtido para análise posterior.



```
def train_network(hyperparameters, train_x, train_y, val_x, val_y):
    learning_rate, dropout_rate = hyperparameters
    im_size = train_x.shape[1]

    # Definindo o modelo da rede neural
    model = Sequential([
        Conv2D(64, (3, 3), activation='relu', input_shape=(im_size, im_size, 3)),
        MaxPooling2D((2, 2)),
        Flatten(),
        Dropout(dropout_rate),  # Usar a variável desempacotada
        Dense(len(street_types), activation='softmax')
    ])

    model.compile(optimizer=Adam(learning_rate=learning_rate),
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])

    model.fit(train_x,
              train_y,
              epochs=5,
              validation_data=(val_x, val_y),
              verbose=0)

    validation_loss, validation_accuracy = model.evaluate(val_x, val_y, verbose=0)
    return validation_loss
```



```
# Defina a função de fitness que será usada pelo GWO
def fitness_function(hyperparameters):
    loss = train_network(hyperparameters, train_x, train_y, val_x, val_y)
    print(f"Particle: {hyperparameters}, Loss: {loss}")
    return loss
```



```
# Parâmetros do GWO
n_agents = 5
n_iterations = 5
dimensions = 2
lower_bound = [0.0001, 0.0]  # Limites inferiores para taxa de aprendizado e dropo
upper_bound = [0.1, 0.5]     # Limites superiores para taxa de aprendizado e dropo

# Inicializar e executar o otimizador GWO
optimizer = gwo(n_agents,
                fitness_function,
                lower_bound,
                upper_bound,
                dimensions,
                n_iterations)

# Desempacotando os melhores hiperparâmetros encontrados
best_learning_rate, best_dropout_rate = optimizer.get_Gbest()
```

```
final_model = Sequential([
    Conv2D(64, (3, 3), activation='relu', input_shape=(im_size, im_size, 3)),
    MaxPooling2D((2, 2)),

    Flatten(),

    Dropout(best_dropout_rate),

    Dense(len(street_types), activation='softmax')
])

final_model.compile(optimizer=Adam(best_learning_rate),
                    loss='sparse_categorical_crossentropy',
                    metrics=['accuracy'])

history = final_model.fit(train_x,
                          train_y,
                          epochs=5,
                          validation_data=(val_x, val_y))
```

4.5. Análise de resultados (Gray Wolf Optimization)

```
Best learning rate: 0.00012542080373561626
Best dropout rate: 0.10106120443028664

Test accuracy: 91.67%

Confusion Matrix:
[[176   8  16]
 [ 14 184   2]
 [  9   1 190]]

Classification Report:
              precision    recall  f1-score   support

       clean       0.88      0.88      0.88       200
      litter       0.95      0.92      0.94       200
     recycle       0.91      0.95      0.93       200

    accuracy                           0.92       600
   macro avg       0.92      0.92      0.92       600
weighted avg       0.92      0.92      0.92       600


AUC for class clean: 0.97
AUC for class litter: 0.98
AUC for class recycle: 0.99
```

Os resultados da otimização com o algoritmo Gray Wolf Optimization (GWO) indicam que foi alcançada uma taxa de aprendizagem ótima de aproximadamente 0.000125 e uma taxa de dropout ótima de cerca de 0.101. Com esses hiperparâmetros, o modelo de rede neural alcançou uma precisão de teste de 91.67%.

Analisando a matriz de confusão, o modelo teve um desempenho equilibrado para a classe 'clean', com valores iguais de precisão e recall, ambos com 88%. A classe 'litter' teve a melhor precisão com 95%, enquanto 'recycle' teve um excelente recall de 95%. Isso sugere que o modelo foi ligeiramente melhor na identificação correta de 'litter' e na cobertura de todos os exemplos verdadeiros de 'recycle'.

No relatório de classificação, as pontuações de F1-score refletem um desempenho sólido e equilibrado em todas as classes, com destaque para 'litter' e 'recycle', que tiveram os maiores valores, indicando uma forte concordância entre precisão e recall.

As pontuações AUC para cada classe são muito altas (0.97 para 'clean', 0.98 para 'litter' e 0.99 para 'recycle'), o que mostra que o modelo tem uma excelente capacidade de distinguir entre as classes positivas e negativas em todas as categorias.

Em resumo, a aplicação do GWO resultou em um modelo altamente eficaz, com hiperparâmetros bem ajustados, que proporcionou um desempenho de classificação robusto.

5. Conclusão e discussão de resultados

A análise dos resultados das otimizações realizadas com os algoritmos Gray Wolf Optimization (GWO) e Particle Swarm Optimization (PSO) revela que ambos foram eficazes na afinação dos hiperparâmetros para uma rede neural, resultando em modelos com alta precisão de teste. O GWO alcançou uma precisão de 91.67%, enquanto o PSO atingiu uma precisão ligeiramente superior de 92.50%. Esta diferença pode ser atribuída às nuances dos processos de busca dos algoritmos e às características específicas do dataset.

A precisão, recall e F1-score demonstram que ambos os modelos tiveram um desempenho equilibrado em todas as classes, com a classe 'recycle' destacando-se em ambas as otimizações. No entanto, 'litter' apresentou a melhor precisão na otimização com GWO, enquanto 'recycle' teve a melhor precisão com o PSO. Isto sugere que diferentes configurações de hiperparâmetros podem favorecer diferentes classes.

As pontuações AUC foram excepcionalmente altas para ambas as otimizações, o que indica uma capacidade consistente dos modelos em distinguir entre as classes positivas e negativas. A robustez do modelo otimizado pelo GWO é confirmada pelas pontuações AUC de 0.97 a 0.99, enquanto o modelo otimizado pelo PSO mostra um desempenho comparável.

Em conclusão, os resultados obtidos indicam que tanto o GWO quanto o PSO são abordagens confiáveis para otimizar os hiperparâmetros de redes neurais em tarefas de classificação. Embora o PSO tenha levado a uma precisão de teste ligeiramente superior, a diferença não é substancial, sugerindo que a escolha entre os dois algoritmos pode depender de preferências ou restrições específicas do cenário de aplicação. Ambos os modelos mostraram um desempenho equilibrado e robusto, validando a eficácia dos algoritmos de otimização de enxame em problemas complexos de classificação.

6. Referências

- SwarmPackagesPy (https://github.com/SISDevelop/SwarmPackagePy);
- ComputaçãoSwarm (https://pt.wikipedia.org/wiki/Intelig%C3%AAncia_de_enxame)
- Ackley's Function (https://en.wikipedia.org/wiki/Ackley_function)